Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Fast Compositing for Cluster-Parallel Rendering

M. Makhinya[†1], S. Eilemann[2,1], R. Pajarola[1]

[1]Visualization and MultiMedia Lab, University of Zürich [2]Eyescale Software GmbH

Abstract

The image compositing stages in cluster-parallel rendering for gathering and combining partial rendering results into a final display frame are fundamentally limited by node-to-node image throughput. Therefore, efficient image coding, compression and transmission must be considered to minimize that bottleneck. This paper studies the different performance limiting factors such as image representation, region-of-interest detection and fast image compression. Additionally, we show improved compositing performance using lossy YUV subsampling and we propose a novel fast region-of-interest detection algorithm that can improve in particular sort-last parallel rendering.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems—Distributed Graphics; I.3.m [Computer Graphics]: Miscellaneous—Parallel Rendering

1. Introduction

Task decomposition for parallel rendering can take place in different stages of the rendering pipeline. Frame-based approaches distribute entire frames, i.e. by time-multiplexing (DPlex) or screen-space decomposition (2D), to different rendering processes. In this cases the compositing stage is fairly straightforward and consists of the collection of frames or tiles on the display destination, and network transmission cost can generally be ameliorated with fairly simple methods.

With respect to Molnar [MCEF94] on the sorting stage in parallel rendering, we can identify three categories of intraframe-like decomposition modes: Sort-first (2D) decomposition divides a single frame spatially and assigns the resulting tiles to the render processes; Sort-last (DB) decomposition does a domain decomposition of the database across the rendering processes. A sort-middle approach can typically not be modified or custom implemented as one needs to intercept the transformed and projected geometry (in scan-space) after primitive assembly which does not translate to a feasible and scalable cluster-parallel rendering system.

For sort-last rendering, the recomposition of the partial frames into the final frame is more time-consuming than for sort-first. A number of special-purpose image compositing hardware solutions for sort-last parallel rendering have been developed. Proposed hardware architectures include Sepia [MHS99], Sepia 2 [LMS*01], Lightning 2 [SEP*01], Metabuffer [BBFZ00], MPC Compositor [MOM*01], PixelFlow [MEP92] and network processing [PMD*07], of which only a few have reached the commercial product stage (i.e. Sepia 2, MPC and PixelFlow). However, the inherent inflexibility and setup overhead have limited their usefulness and with the recent advances in CPU-GPU interfaces, combinations of software and GPU-based solutions offer more flexibility at comparable performance and lower price.

Distributed cluster-parallel image compositing, even if only for image assembly as in sort-first rendering, is fundamentally limited by the network throughput that bounds the amount of image data that can be exchanged between nodes. Hence efficient image coding, compression and transmission techniques must be considered in this context. In this paper we study in detail different performance limiting factors such as image formats, pixel read-back, region-of-interest selection and GPU assisted compression.

The contributions presented in this paper not only consist of an experimental analysis about the impact of image throughput on parallel compositing but also introduce a novel region-of-interest (ROI) identification algorithm. We furthermore show how image throughput can be improved using YUV subsampling combined with specific RLE methods as well as applying an effective ROI selection method.

† makhinya@ifi.uzh.ch, eile@eyescale.ch, pajarola@acm.org

2. Related Work

A number of general parallel rendering concepts have been discussed before, such as parallel rendering architectures, distributed compositing, load balancing, data distribution, or scalability. However, only a few generic APIs and parallel rendering systems exist which include VR Juggler [BJH*01] (and its derivatives), Chromium [HHN*02], OpenGL Multipipe SDK [BRE05] and Equalizer [EMP09], of which the last is used for the work presented in this paper.

For sort-last rendering, a number of parallel compositing algorithm improvements have been proposed in [MPHK94, LRN96, SML*03, EP07, YWM08, PGR*09]. In this work, however, we focus on improving the image compositing data throughput by data reduction techniques applicable to different parallel compositing methods. Hence we consider the two extreme cases of serial and binary-swap or direct-send compositing in our experiments, having $O(N)$ serially or exactly two full images concurrently to exchange in total between the N rendering nodes respectively. For sort-first parallel rendering, the total image data exchange load is fairly simple and approaches $O(1)$ for larger N.

To reduce transmission cost of pixel data, image compression [AP98, YYC01, TIH03, SKN04] and screen-space bounding rectangles [MPHK94, LRN96, YYC01] have been proposed. However, with modern GPUs these concepts do not always translate to improved parallel rendering as increased screen resolutions and faster geometry throughput impose stronger limits under which circumstances these techniques are still useful. A disproportional growth and shift in compositing-cost that increases with the number of parallel nodes can in fact negatively impact the overall performance, as we show in our experiments. Therefore, care has to be taken when applying image compression or other data reduction techniques in the image compositing stage of parallel rendering systems.

3. Parallel Rendering Framework

The parallel rendering framework of our choice, i.e. Equalizer [EMP09], has a number of advantages over other systems, in particular its scalability and flexibility of task decompositions. However, the basic principles of parallel rendering are similar for most approaches, and the analysis, experiments and improvements on image compositing presented in this paper are generally applicable.

In a distributed rendering setting, the general execution flow is as follows, omitting event handling and other application tasks: *clear, draw, read-back, transmit* and *depth-based composite* or *2D-assemble* for display. Clear and draw are largely ignored in this work, but we run experiments with different geometric complexities that affect the draw speed to gain insight into performance behavior under different configurations.

In the following we focus on the read-back, transmission and compositing stages and in particular on the image processing throughput which, for a given network bandwidth, is chiefly affected by the image representation.

4. Distributed Image Compositing

Distributed parallel image compositing cost is directly dependent on how much data has to be sent over the network, which in turn is related to how much screen space is actively covered. Additionally, read-back and transmission times are also affected by image color and compression formats.

In the following we first introduce our generic sparse-image representation approach that can be used in any sort-first or sort-last parallel rendering configuration. Further data reduction can be achieved using image compression. However, this must meet demanding requirements, as its overhead has to be strictly smaller than any transmission gainings, which can be difficult to achieve.

4.1. Regions-of-Interest

In sort-last rendering, every node potentially renders into the entire frame buffer. With an increasing number of nodes the set of affected pixels typically decreases, leaving blank areas that can be omitted for transmission and compositing.

Our region-of-interest (ROI) algorithm splits the frame buffer into parts with active pixels and excluded blank areas. The active ROI is less or equal to the frame buffer size and depends on how many pixels have actually been generated. Thus the maximal benefit will be reached when each node renders only to a compact region in the frame buffer. This assumption largely holds for hierarchically structured data which is often used to accelerate culling and rendering.

The ROI algorithm is called when all rendering is finished right before read-back. Identified subregions are individually treated for read-back, compression and transmission (RBCT) as well as compositing, any of which is reused from the original parallel rendering framework.

4.1.1. ROI selection

For an efficient RBCT process, there are several desired features that final regions should exhibit:

- compact rectangular shape
- coverage of all generated pixels in the frame buffer
- no region overlap
- smallest possible area for limited number of regions

Since the number of regions must be limited to avoid excessive GPU read-back requests, the last feature is the most difficult to achieve. Our proposed ROI method reformulates the last criterium as follows: we aim to exclude as much of the blank frame buffer area as possible with as little effort as possible. Our solution preserves the other criteria while effectively removing the undesired blank spaces.

ROI selection itself requires time and if its area is not significantly smaller than the entire frame buffer it may affect performance negatively due to overhead. The decision

whether to use ROIs or not is made dynamically while identifying blank regions. If the blank area is too small with respect to the frame buffer size, ROI selection *fails* and is turned off, delayed for one frame, meaning the whole screen will be read-back in the next frame. Every consecutive ROI failure causes an increasing delay to the next ROI estimation up to a certain limit at which ROI estimation is done at a regular periodic rate. Currently this limit is at 64 frames, equalling to a few seconds of real-time interaction and minimizing overhead of failed ROI estimation.

The algorithm itself consists of two main parts: per-block occupancy calculation performed on the GPU and region split estimation executed on the CPU as indicated in Figure 1.

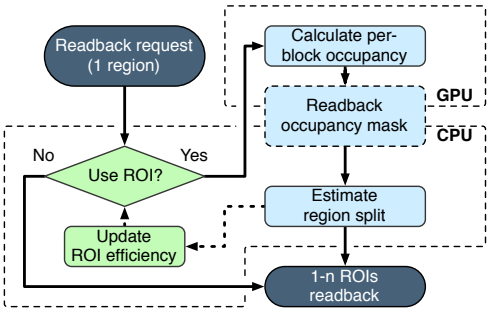


Figure 1: *ROI selection algorithm.*

To speed up the region split process, a block-based occupancy mask is used. This mask is computed using a fragment shader on the GPU. It contains a flag for each grid block indicating whether it contains rendered pixels or not. For ROI selection, the reduced-size bitmask is transferred to the main memory. On the CPU a block-based region split is computed aligning the ROIs to the regular grid blocks. If enabled, the necessary ROIs will then be read back from the GPU for further compression, transmission and final compositing. Empty areas are dedected using either a specified solid background color or by analyzing z-buffer values. 16×16 blocks are used for the occupancy mask, which provides a good trade-off between speed and precision.

The split algorithm is based on recursive region subdivision. Each iteration consists of two steps: finding the largest rectangular blank area (a hole); and best split determination based on hole position and size. Figure 2 illustrates the recursive per-block occupancy hole detection and split process. Depending on a hole position within a region, there are several possible split options. For this particular hole's configuration, there are two ways to obtain rectangular sub-areas that do not include the hole itself but only the rest of the image, one of these is shown in each subsequent step.

4.1.2. Hole search

Identifying the largest unused rectangular region within the block-based occupancy mask is efficiently performed using a *summed area table* (SAT). The occupancy map is

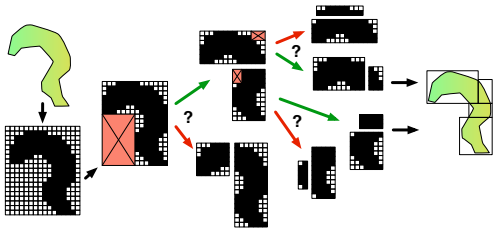


Figure 2: *Recursive largest hole detection and subtraction for recursive ROI selection.*

transformed into a SAT with entries indicating the number of empty blocks up to the given index. Thus emptiness of any rectangular region can quickly be verified by four SAT lookups and comparison to the block size of the region.

A maximal empty-area search algorithm is defined as a sequence of requests to the SAT, and executed in the following way over the block-based occupancy map SAT:

1. Search in scan-line order bottom-up for empty blocks.
2. In each step, find the *intermediate largest hole*, a rectangular hole that includes the current block and is bounded by the upper right corner of the map.
3. Update the current largest hole if the new one is bigger.

The *intermediate largest hole* search in Step 2. is based on a three-fold region growing strategy as shown in Figure 3. The strategy is to first grow an empty square region diagonally, followed by growing a tall empty rectangle vertically with subsequently reduced width on every empty-region test failure. The same is then performed analogously horizontally. In each growth step an empty-region test corresponds to a query of the SAT occupancy map.

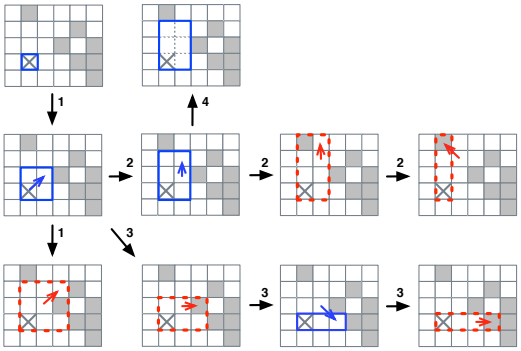


Figure 3: *Largest hole searching first proceeding diagonally upwards (1), then vertically with decreasing width (2) and last horizontally with decreasing height (3). Eventually the largest hole found (4) is reported.*

To avoid identifying too small empty areas, the hole search procedure is terminated if either a minimal absolute or relative empty region size threshold is not reached. In that case a zero-sized hole is reported to avoid further recursion. These values can be set to quite high values (e.g. 200 blocks and 2%) and still give acceptable cuts.

4.1.3. Split estimation

A region split is executed once after the largest hole in a current frame buffer region has been found. There are four categories of hole positions as shown in Figure 4, of which only the first one leads to a simple split into two new vertical or horizontal rectangular regions.

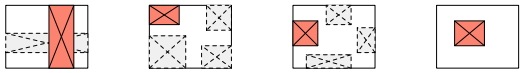


Figure 4: *Different categories of hole positions: (a) through, (b) corner, (c) side and (d) center.*

The symmetry classes of possible region splits are shown in Figure 5. For a corner hole only two variants are possible, with one either vertical or horizontal line aligned to one of the hole's edges. A side hole has four different options, and a center hole has two unique configurations as well as four which reduce it to a side hole.

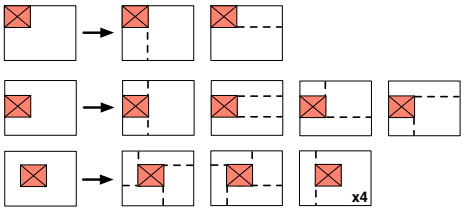


Figure 5: *Symmetry categories of region splits.*

To find the best split, our algorithm maximizes the area that can be removed in the next subdivision. That is, a hole search is performed for the subregions of every possible split and the accumulated size of all holes is considered. For the best split the determined hole positions are forwarded to the recursive split processes for each subregion.

In order to avoid excessive and repeated hole searches, information from common subregions is shared. There is a maximum of 16 different subregions that have to be checked depending on the hole position, as depicted in Figure 6. For corner and side holes, disappearing subregions are considered to have a zero hole area.

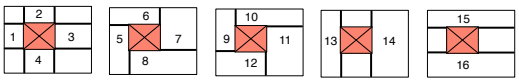


Figure 6: *Different subregions for split calculation.*

4.2. Image Compression

Basic run-length encoding (RLE) has been used as a fast standard to improve network throughput for interactive image transmission. However, it only gives sufficient results in specific rendering contexts and fails to provide a general improvement as will be shown also in our experimental results. RLE only works to compact large empty or uniform color areas but is often useless for non-trivial full frame color results. We analyze two enhancements to improve this, per-component RLE compression and *swizzling* of color bits.

More complex (and lossy) image compression techniques (e.g. such as LZO or EZW) may promise better data reduction, however, at the expense of significantly increased compression cost which renders many solutions infeasible in this context (see Appendix A). Additionally, lossy compression (e.g. such as used in VNC) may only be tolerable when compression artifacts are masked by motion and high frame rates. In this paper we study the benefit of YUV subsampling, also combined with RLE, as it can provide very fast and effective compression for scenes in motion, and lossless reconstruction can easily be incorporated by incremental transmission of the missing data from the last frame. Hence we focus on a few provenly simple and very fast techniques such as RLE and YUV subsampling, see also Section 6 for some more discussion.

4.2.1. Run-length encoding

For the basic RLE method we use a fast 64-bit version that compares two pixels at the same time (8-bit per channel RGBA format). While this method is very fast it shows poor compression results in most practical settings. A general concept in image compression is to treat color components separately, as illustrated in Figure 7. Results on the different RLE schemes are reported in the experiments.

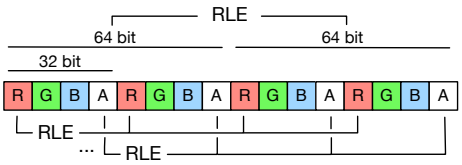


Figure 7: *Comparison of 64-bit and per-component RLE.*

A second improvement is *bit-swizzling* of color values before per-component compression. That way the bits are reordered and interleaved as shown in Figure 8. Now per-component RLE compression separately compresses the higher, medium and lower order bits, thus achieving stronger compression for smoothly changing color values.

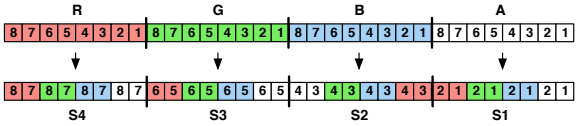


Figure 8: *Swizzling scheme for reordering bits of 32-bit RGBA values.*

4.2.2. YUV subsampling

Lossy compression in our study consists of RGB to YUV color transformation and chroma subsampling (4:2:0) since it allows fast computation, good general compression and incremental reconstruction to full chroma color if necessary. Without an additional RLE stage, a compression ratio of 2 : 1

can always be achieved this way with good image quality, especially for dynamic motion. Color transformation, subsampling and byte-packing can all be done efficiently in a fragment shader such that not only network transmission but also read-back from the frame buffer will be improved.

Figure 9 illustrates the YUV subsampling and byte packing. While luminosity values are packed to the left of a 2×2 4-channel pixel block, the chromaticity values are averaged. However, to reduce color distortion at silhouettes only non-zero, non-background color values are averaged. Alpha values are pair-wise averaged on a scan line, and are not needed in the case of sort-first rendering. An example of chroma subsampling is given in Section 5.4.

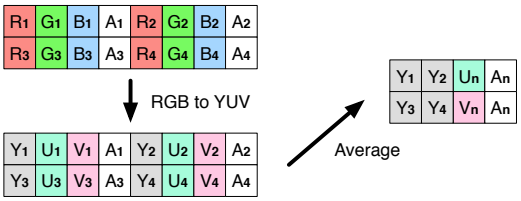


Figure 9: *Lossy RGB to YUV transform and subsampling.*

The particular color sampling and packing pattern has been chosen to easily support subsequent RLE compression. The above outlined RLE method processes pixels in scanline order and thus component-wise RLE can directly be applied after YUV transformation and subsampling.

5. Experimental Analysis

Any parallel rendering system is fundamentally limited by two factors: the rendering itself, which includes transformation, shading and illumination; as well as compositing multiple partial rendering results into a final display, i.e. image. While an exceeding task load of the former is the major cause for parallelization in the first place, the latter is often a bottleneck due to limited image data throughput.

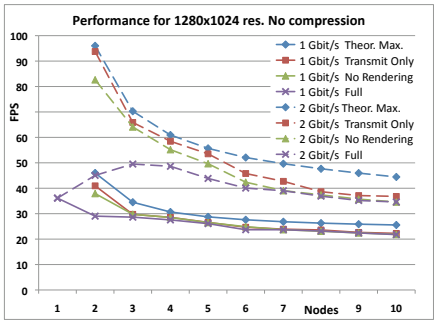
In our experiments we investigate the image transmission and compositing throughput of sort-first and sort-last parallel rendering. For the sort-last we study serial as well as direct-send (DS) compositing, as they exhibit two extreme cases in parallel compositing strategies in terms of image transmission.

All tests were carried out on a 10-node cluster, *Hactar*, with the following technical node specifications: dual 2.2GHz AMD Opteron CPUs; 4GB of RAM; one or two GeForce 9800 GX2 GPUs and a high-resolution 2560×1600 pixel LCD panel; 2 Gbit/s Myrinet and switch, as well as 1 Gbit/s ethernet network.

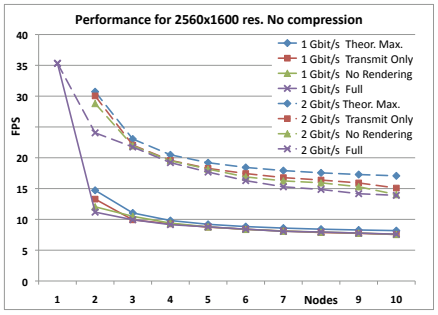
5.1. Throughput Limits

First, we identify the typical distributed image compositing and thus parallel rendering performance limits when image compression is not used. This will also demonstrate that the used parallel rendering framework is in fact very resource efficient and achieves the expected limits.

Figure 10 shows different image throughput limits for two different frame buffer resolutions, without using image compression, in the context of sort-first rendering. In sort-first rendering, given N rendering nodes (one of which is also the display node), the network will be loaded with the cost of transmitting $\frac{N-1}{N}$ of the full frame buffer image data to the final display node. Hence with increasing N, the image transmission throughput and thus compositing speed is expected to decrease. In the limit ($N \to \infty$) we can expect a maximal frame rate of μ / s_{FB} for a given network bandwidth μ and frame buffer size s_{FB}. Using *netperf*, we evaluated a realistic achievable data transmission rate of $\mu = 115MB/s$ and $\mu \leq 240MB/s$ for 1Gbit and 2Gbit networks. Thus for a frame buffer size s_{FB} of 1280×1024 ($5MB$) we expect up to 23fps or 48fps, and for 2560×1600 ($16MB$) 7fps or 15fps respectively for the different network speeds. The maximal achievable frame rates limited by the bandwidth as described above are indicated in Figure 10 with Theor. Max..



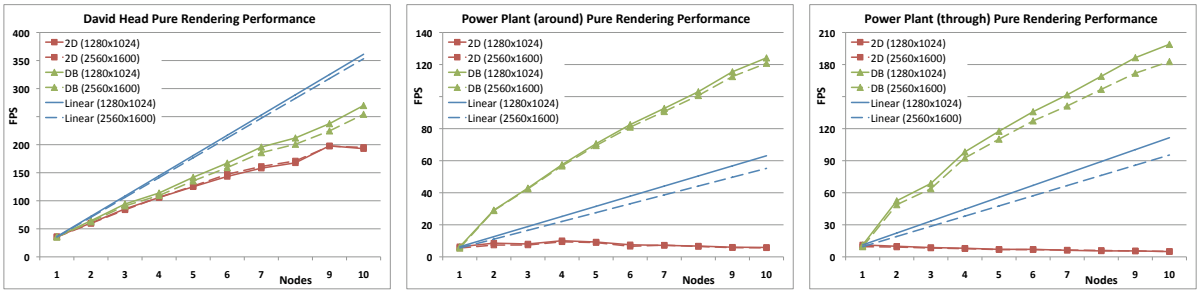
(a)



(b)

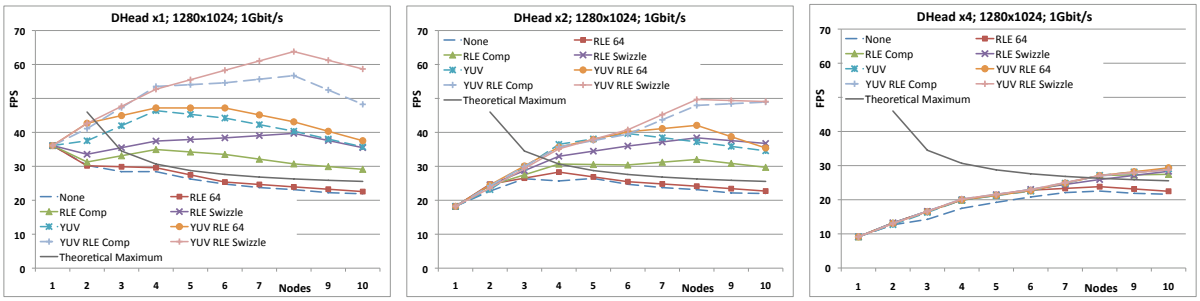
Figure 10: *Maximal theoretical and real image throughputs for (a) 1280x1024px and (b) 2560x1600px resolutions.*

Ignoring any rendering cost but only focusing on image throughput and compositing, the experiments show that our parallel rendering setup is efficient and approaches the expected limits. The Transmit Only graphs in Figure 10 indicate the system's limit simply for transmitting the partial frame buffer results to the destination, which nicely follows the expected limits in particular for the larger frame buffer. The No



(a) (b) (c)

Figure 11: *Rendering-only performance: (a) David Head, (b) Power Plant (fly around), and (c) Power Plant (fly through).*



(a) (b) (c)

Figure 12: *Different color compression for David Head model: (a) normal rendering speed, (b) double rendering load, (c) four times rendering load (same model rendered one, two and four times respectively). Theoretical maximum lines correspond to the best speed possible when only considering the time required for uncompressed image transmission over the given network.*

Rendering curves for the entire sort-first compositing task, but still not accounting for actual rendering itself, indicate the hard limits of the rendering system if no image compression is used. These results still closely follow the bandwidth constraints and theoretical expected limits, and thus demonstrate that the sort-first compositing stage does not introduce any significant overhead.

A full rendering test, labeled as Full in Figure 10, with only a small polygonal object further confirms the limits and resource efficiency of the system. The curves show that the frame rate is quickly limited by the image throughput and decreases accordingly for larger N. Only for a small frame buffer and slow network configuration the frame rate initially increases when adding rendering nodes until being dominated by the image throughput constraints. Hence apparently there is no notable overhead introduced in the parallel rendering system.

Basic scalability for larger models is confirmed in Figure 11, showing what can be reached in the best case just from rendering, this time not taking any image transmission and compositing into account. Both sort-first and sort-last parallel rendering improve rendering speed almost linearly for uniformly distributed geometry (Figure 11(a)), However, sort-first scalability starts to flatten out at some point as expected due to smaller viewports but constant per-node culling costs. For the large Power Plant model, the results in Figs. 11(b) and 11(c) show that sort-last rendering can scale superlinearly due to GPU caching effects. It also shows that for uneven distributed geometry, a regular sort-first screen decomposition cannot improve rendering speed, which is expected as well.

5.2. Sort-first Performance

To evaluate compression benefits in relation to the basic image throughput we have analyzed RLE, per-component RLE, swizzle RLE and YUV subsampling. While YUV is lossy, it can often be used without noticeable loss in visual quality (see also Figure 17), and it can be combined with RLE. Figure 12 shows the overall frame rate due to image compression for varying geometric model complexity. It shows that basic RLE does not help for non-uniform color images. While per-component or swizzle RLE are more costly, they can achieve an improvement. Swizzle RLE works reasonably well as it can improve image transmission more significantly.

YUV subsampling reduces the chromatic color components by a factor of 4 and thus the total image size by 2 at minimal extra image processing cost. This data reduction shows immediate effects on the frame rate as shown in Figs. 12(a) and 12(b). It is also confirmed that basic RLE

does not improve upon YUV (compare curves YUV with YUV RLE 64). However, per-component or swizzle RLE based compression on top of YUV subsampling can further improve the image throughput and overall frame rate (see curves YUV RLE Comp with YUV RLE Swizzle).

Figs. 12(b) and 12(c) show the effect of pure rendering limits for larger models and consequently achievable parallel speedup. If the 3D data complexity is so high that rendering itself is the bottleneck, adding more nodes improves sort-first parallel rendering until the image throughput limit is reached (see curves None with RLE 64). Only after that intersection point, image compression has a noticeable effect.

Figure 12 furthermore confirms the 23fps limit for $N \to \infty$, with $s_{FB} = 1280 \times 1024$ screen and bandwidth $\mu = 115MB/s$ since all measures converge to that, unless compression is applied. These results will scale appropriately with screen resolution and network bandwidth.

The complete composing performance is defined by:

1. Read-back
2. Compression (if compression is used)
3. Transmit
4. Decompression (if compression is used)
5. Per-image compositing

The results in Figure 12 help to further understand the influence of different approaches to sort-first rendering.

In the tests above the per-vertex colored David Head model was placed in the center and nearly covering the full screen. The animation rotated the model around x and y. The results confirm that image transmission has the most significant influence compared to read-back and compositing which cause very little overhead. Basic RLE compression proves to have poor performance and only swizzle RLE can sufficiently compensate extra compression cost with increased image throughput, outperforming in total other RLE versions.

YUV subsampling alone improves performance due to the fixed data reduction, which can further be improved in combination with per-component or swizzle RLE.

RLE compression is implemented using OpenMP on the CPU, however, our parallel framework is already running four threads and the CPUs are fully used. If more than four cores are used, one could expect improved performance of RLE (we observe doubling of RLE speed on 2 cores, when tested without the rest of the framework running).

5.3. Sort-last Performance

Sort-last parallel polygon rendering uses the z-buffer in order to perform z-depth compositing of partial rendering results. Thus the z-depth buffer data also has to be sent over the network.

5.3.1. Depth component compression

To determine the best depth component compression we have set color compression to RLE and are using serial sort-last compositing which imposes a network image transmission load proportional to the number N of rendering nodes. We use color RLE since it removes blank screen space effectively which is typical for sort-last rendering. In Figure 13 uncompressed depth data is indicated in the graph by None. Measures are shown for different data models (David Head, Power Plant around and fly-through), network bandwidth and for $N = 2, 6, 10$ nodes.

For uniform partitioning of geometry each node is rendering $1/N$ of the data and less screen space is covered with increasing N for sort-last rendering. Hence empty-pixel skipping is more important than actual depth-value compression to transmit the full-frame sized depth-buffer to the final destination node. This is supported in Figure 13 with simple RLE depth compression performing better than the more complex per-component or swizzle variants.

However, in the case where partitioning of the data does not lead to sparse images for sort-last compositing, but most of the frame buffer is covered, per-component RLE compression shows better results. This experiment is reported in the last two columns of Figure 13 where the model is rendered twice on two nodes, covering the entire frame buffer on both nodes. In Section 6 we briefly provide some more discussion also on GPU usage.

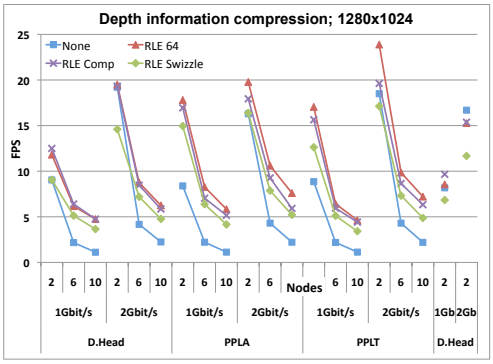
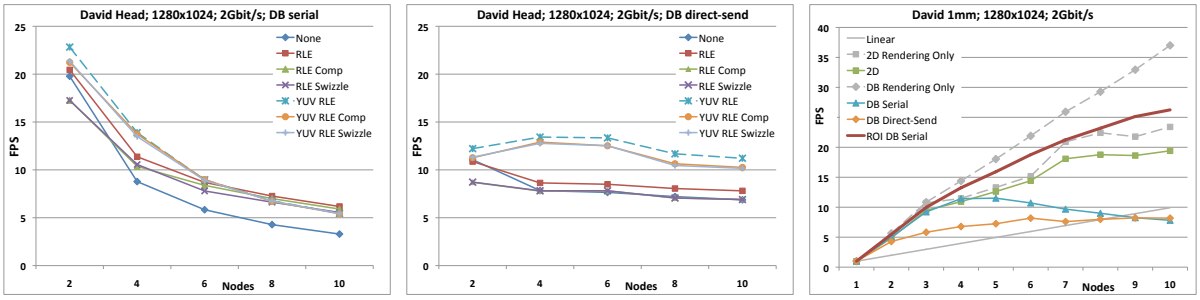


Figure 13: *Image throughput performance comparison of different depth-component compression methods.*

5.3.2. Color component compression

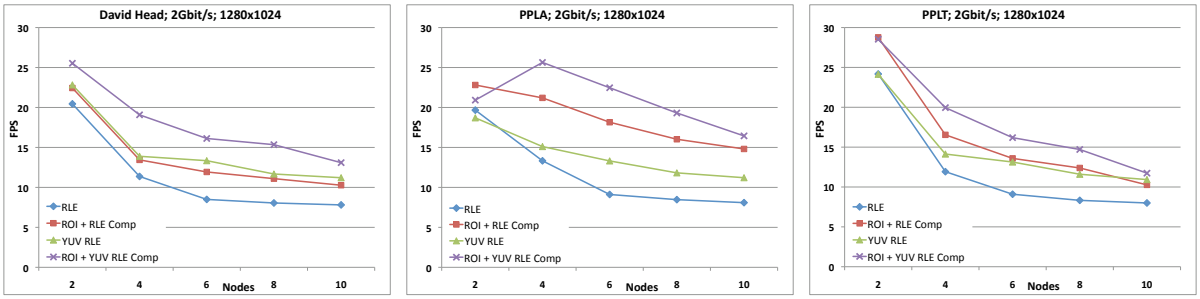
To determine the best color compression we have fixed depth compression to RLE. As observed before, the images generated from sort-last rendering are likely to have large empty regions with increasing N. When using direct-send compositing [SML*03, EP07], each node renders a part of the 3D data into a full-sized frame buffer, followed by depth-compositing of a part of the entire viewport. The composited sub-regions are eventually sent to the destination node for final assembly.

Figure 14(a) shows that simple RLE compression is effective for color if increasingly larger parts of the partially rendered images are empty. YUV subsampling has much less effect for such sparse image data. Almost identical results



(a) (b) (c)

Figure 14: *Image throughput comparison of different color compression methods for (a) serial and (b) direct-send sort-last compositing. (c) ROI selection impact on sort-first (2D) as well as sort-last (DB) serial and direct-send compositing modes.*



(a) (b) (c)

Figure 15: *Effect of our ROI method on image throughput. (a) David Head, (b) Power Plant (fly around) and (c) Power Plant (fly through) model.*

were found for rendering the Power Plant model, but are omitted here due to space.

In contrast to serial compositing, direct-send exhibits a constant amount of image data that has to be transmitted between the parallel rendering nodes, which is also evidenced in Figure 14(b). Due to the split-frame compositing and different distribution of rendering and pixel data, YUV subsampling can notably improve image throughput here.

The results suggest that for a small number of nodes, depending on the network bandwidth, serial compositing performs better than direct-send. Direct-send will be beneficial for a larger number of rendering nodes and large complex 3D data sets.

5.3.3. ROI compression

So far we can conclude that for sort-last rendering the efficient encoding of the sparse image data is important, i.e. removal of blank pixel data. Furthermore, the experiments on sort-first rendering have shown that for large non-uniform color regions YUV subsampling and optionally swizzle RLE coding achieve significant improvements in image throughput. In the following we test the influence of our ROI selection algorithm. Serial compositing up to 5 nodes and afterwards direct-send compositing is used as a reference point, when ROI is executed in serial fashion only. RLE compression used for simple tests, and RLE per-component for ROI tests (since blank areas are already excluded).

Figure 15 shows that our ROI algorithm clearly outperforms RLE for empty pixel removal. Despite the additional cost to detect ROIs (from 0.79ms to 2.5ms for 1280×1024) and splitting an image into multiple regions, our ROI selection algorithm quickly and effectively identifies and removes blank frame buffer areas. In combination with YUV subsampling further significant speedups can be achieved. The decrease of frame rates is due to the fundamental image throughput limits outlined at earlier.

5.3.4. ROI scalability

Without optimization of the image throughput, scalability is quickly limited due to the distributed image compositing stage, i.e. network transmission as reported e.g. in [EMP09]. Below we demonstrate the positive impact of ROI selection on the compositing stage. Experiments are conducted with the large David 1mm model displayed horizontally. For sort-first rendering the screen was divided into N equal vertical tiles, for sort-last the data is split into N equal chunks.

In Figure 14(c) the theoretically possible performance is indicated with Rendering only for sort-first (2D) and sort-last rendering (DB), without accounting for any image transmission and compositing. The actually achieved frame rates of

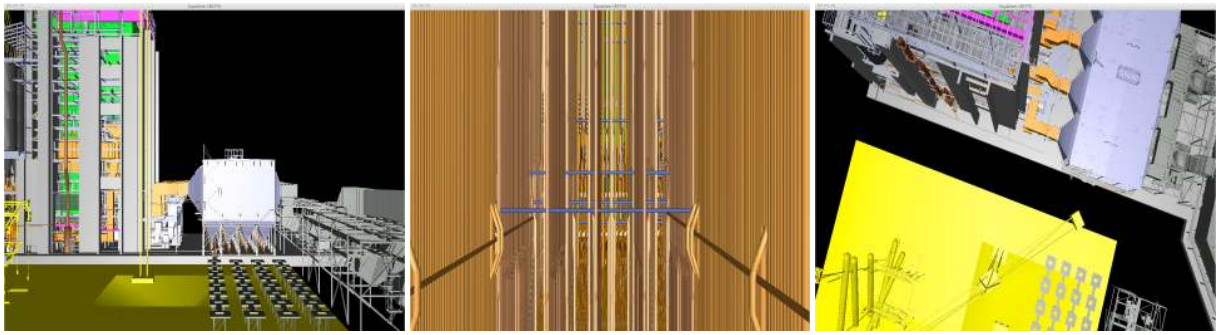


Figure 16: *Screen shots along the Power Plant fly-through path.*

sort-last rendering are shown for serial and direct-send compositing. Superlinear performance of rendering can be explained by caching effects in main memory and GPU. The negative performance impact of the limited image throughput is obvious as the speedup quickly approaches the expected frame rates and flattens out. In contrast, the ROI enhanced sort-last rendering keeps up scalability much longer and performs superlinearly up to the maximum number of nodes tested.

While sort-first rendering in general shows less impact due to image throughput limits than sort-last methods for large complex 3D data, ROI enhanced sort-last rendering nevertheless outperforms it considerably.

5.4. View Examples

In Figure 17 we show the typical configuration for rendering the David Head model fully covering the frame buffer. Sort-last decomposition of the data is indicated by color coding and sort-first decomposition by the vertical tiles (actual colors of David Head model, used in the testing, are not shown). Also we demonstrate the YUV chroma subsampling in the lower half of the image, exhibiting visual artifacts at discontinuous color boundaries but only if the image is magnified significantly. Additionally the fly-through path for the Power Plant model is indicated with example screen shots along that path given in Figure 16.

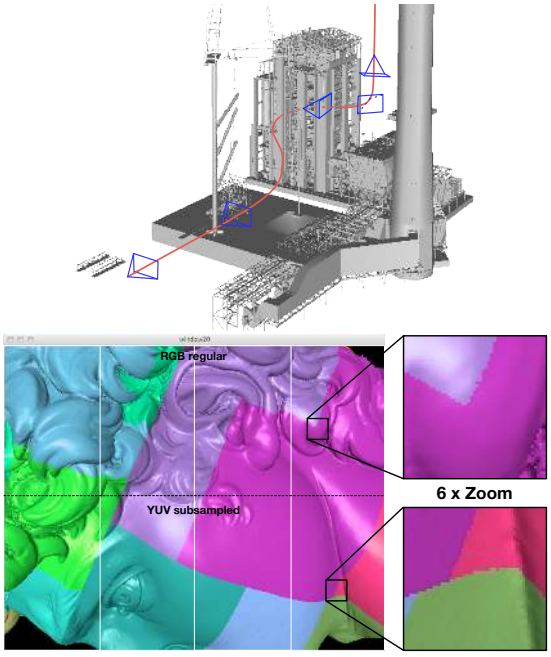


Figure 17: *Typical view of the Power Plant fly-through path and the full-screen David Head model.*

6. Conclusions and Future Work

The analysis and experiments presented in this paper highlight the impact of the image throughput on the distributed compositing stage of cluster-parallel rendering systems. We furthermore demonstrate the potential improvements using image compression and ROI selection techniques, for which we introduce novel and fast algorithms.

Sort-first parallel rendering is fundamentally limited by a very strict network bandwidth constraint that can only be alleviated using frame buffer data reduction which can be carried out at very high frame rates on any input image. For sort-last rendering approaches, the image throughput is also a critical and limiting performance factor independent of the compositing strategy. It has been shown that the typically occurring blank frame buffer areas can better be removed with a clever ROI algorithm than simple RLE for empty-pixel skipping.

As GPU based image compression algorithms become more widely available it will be interesting to incorporate and analyze these in the future. The integration will be more complex as it requires a tighter integration with the parallel rendering framework. The compositing stage will be less orthogonal to the parallelized draw stage as GPU, frame buffer and OpenGL contexts must be shared and accessed directly. Additionally, while CPU image compression can run concurrently to rendering on the GPU, GPU based image compression will steal GPU resources away from rendering which

could result in slower overall frame rates.

Acknowledgments

This work was supported in part by the Swiss National Science Foundation under Grant 200021-116329/1. The authors would like to thank and acknowledge the following institutions and projects for providing 3D test data sets: the Digital Michelangelo Project, Stanford 3D Scanning Repository and the UNC Walkthru Project.

References

[AP98] AHRENS J., PAINTER J.: Efficient sort-last rendering using compression-based image compositing. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization* (1998). 2

[BBFZ00] BLANKE W., BAJAJ C., FUSSEL D., ZHANG X.: *The Metabuffer: A Scalable Multi-Resolution 3-D Graphics System Using Commodity Rendering Engines*. Tech. Rep. TR2000-16, University of Texas at Austin, 2000. 1

[BJH*01] BIERBAUM A., JUST C., HARTLING P., MEINERT K., BAKER A., CRUZ-NEIRA C.: VR Juggler: A virtual platform for virtual reality application development. In *Proceedings of IEEE Virtual Reality* (2001), pp. 89–96. 2

[BRE05] BHANIRAMKA P., ROBERT P. C. D., EILEMANN S.: OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization* (2005), pp. 119–126. 2

[EMP09] EILEMANN S., MAKHINYA M., PAJAROLA R.: Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics* (May/June 2009). 2, 8

[EP07] EILEMANN S., PAJAROLA R.: Direct send compositing for parallel sort-last rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2007). 2, 7

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics 21*, 3 (2002), 693–702. 2

[LMS*01] LOMBEYDA S., MOLL L., SHAND M., BREEN D., HEIRICH A.: Scalable interactive volume rendering using off-the-shelf components. In *Proceedings IEEE Symposium on Parallel and Large Data Visualization and Graphics* (2001), pp. 115–121. 1

[LRN96] LEE T.-Y., RAGHAVENDRA C., NICHOLAS J. B.: Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (July-September 1996), 202–217. 2

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32. 1

[MEP92] MOLNAR S., EYLES J., POULTON J.: PixelFlow: High-speed rendering using image composition. In *Proceedings ACM SIGGRAPH* (1992), pp. 231–240. 1

[MHS99] MOLL L., HEIRICH A., SHAND M.: Sepia: scalable 3D compositing using PCI pamette. In *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines* (1999), pp. 146–155. 1

[MOM*01] MURAKI S., OGATA M., MA K.-L., KOSHIZUKA K., KAJIHARA K., LIU X., NAGANO Y., SHIMOKAWA K.: Next-generation visual supercomputing using PC clusters with volume graphics hardware devices. In *Proceedings ACM/IEEE Conference on Supercomputing* (2001), pp. 51–51. 1

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel volume rendering using binary-swap image composition. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 59–68. 2

[PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W., THAKUR R.: A configurable algorithm for parallel image-compositing applications. In *Proceedings ACM/IEEE Conference on High Performance Networking and Computing* (2009), pp. 1–10. 2

[PMD*07] PUGMIRE D., MONROE L., DAVENPORT C. C., DUBOIS A., DUBOIS D., POOLE S.: NPU-based image compositing in a distributed visualization system. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (July/August 2007), 798–809. 1

[SEP*01] STOLL G., ELDRIDGE M., PATTERSON D., WEBB A., BERMAN S., LEVY R., CAYWOOD C., TAVEIRA M., HUNT S., HANRAHAN P.: Lightning-2: A high-performance display subsystem for PC clusters. In *Proceedings ACM SIGGRAPH* (2001), pp. 141–148. 1

[SKN04] SANO K., KOBAYASHI Y., NAKAMURA T.: Differential coding scheme for efficient parallel image composition on a pc cluster system. *Parallel Computing 30*, 2 (2004), 285–299. 2

[SML*03] STOMPEL A., MA K.-L., LUM E. B., AHRENS J., PATCHETT J.: SLIC: Scheduled linear image compositing for parallel volume rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 33–40. 2, 7

[TIH03] TAKEUCHI A., INO F., HAGIHARA K.: An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Computing 29*, 11-12 (2003), 1745–1762. 2

[YWM08] YU H., WANG C., MA K.-L.: Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings IEEE/ACM Supercomputing* (2008). 2

[YYC01] YANG D.-L., YU J.-C., CHUNG Y.-C.: Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *Journal of Supercomputing 18*, 2 (February 2001), 201–22–. 2

Appendix A

While LZO is considered to be a very fast general compression technique, Figure 18 demonstrates that it performs worse than swizzle RLE when applied in real time rendering.

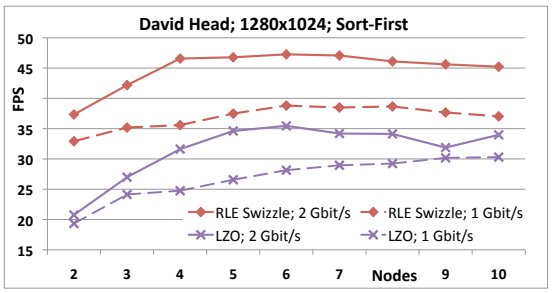


Figure 18: *Comparison between LZO and swizzle RLE compression for sort-first rendering of the David Head model.*